On the Semantics of Message Passing Processes

Lindsay Errington

Dept. of Computing, Imperial College, London SW7 2BZ, U.K.
and
Kestrel Institute, 3260 Hillview Ave., Palo Alto, California 94304, U.S.A.
lindsay@kestrel.edu

Abstract

Let J be a *shape* in some category Shp for which there is a functor $\kappa : \mathsf{Shp} \longrightarrow \mathsf{Cat}$. A *categorical transition system* (or system) is a pair $(J, \kappa(J) \longrightarrow \mathsf{C})$ consisting of a shape labelled by a functor in a category in C.

Systems generalize conventional labelled transition systems. By choosing a suitable universe of shapes, systems can model concurrent and asynchronous computation. By labelling in a category, rather than an alphabet or term algebra, the actions of an algorithm or process can have structure.

We study a class of systems called *twisted systems* having the form $\mathbf{S} = (J, F\widetilde{J} \longrightarrow \mathsf{C})$ where J is a reflexive graph and $\widetilde{(-)} : \mathsf{RGrph} \longrightarrow \mathsf{RGrph}$ is the *twisted graph* construction. The relevance of twisted systems lies in the relationship between twists and spans. A functor $FJ \longrightarrow \mathsf{Sp}(\mathsf{C})$ into a bicategory of spans is equivalent to a functor $F\widetilde{J} \longrightarrow \mathsf{C}$.

The connection with spans means that when the target category $\mathsf{C} = \mathsf{Set}$, then following Burstall, a twisted system can be viewed as a generalized flow-chart. The theory extends to modeling interacting processes. If $\mathbf{U}$ is a system, then a process of type $\mathbf{U}$ is a system $\mathbf{S}$ and a morphism $\mathbf{p} : \mathbf{S} \longrightarrow \mathbf{U}$. The system $\mathbf{U}$ represents the *interface* to the process. It describes what can be observed and what the process offers to the environment for interaction. The system $\mathbf{S}$ describes the internal behaviour of the process and the morphism $\mathbf{p}$ describes how $\mathbf{S}$ realizes observable behaviour. Processes compose by pullback over a common interface.

1 Introduction

Various categories have been used to interpret parallel languages including value-passing and non-value-passing process calculi. These include categories of traces [6,44], trees [30,31,40,23], event structures [42,45] and more recently presheaves [46,12]. In all cases the semantics is denotational in the sense that it is compositional and fixpoints are used to model recursive processes. The use of fixpoints means that the category must be equipped with a suitable

ordering. This usually precludes the use of transition systems with loops. Inevitably, any recursive behaviour is unwound and all the structures mentioned above are either trees or suitably "tree-like".

Following the example set with the introduction of the π-calculus [37], two methods are used for languages with variables and value-passing. In an *early* semantics, all variables are discarded from the outset and terms are translated into non-value-passing form. Variables are assumed to range over a fixed set of values, V, and each variable is instantiated with all possible values. In CCS, for example, a term $a(x).P$ becomes $\sum_{x \in V} a_x.P[a_x/x]$. In a *late* semantics substitution is postponed. A term $a(x).P$ is viewed as a λ abstraction with argument x. Substituting a value for x is performed as part of parallel composition where the term is applied to the value received. Whichever method is used, the effect is that variable assignments are recorded in the branching structure.

The point we wish to make is that, while unfolding loops and expanding variables may lead to simpler models, these operations also lead to a considerable loss of information. In particular, we lose track of the fact that some transitions are instances of the same action in the program. This becomes relevant when the theory is intended to form a basis for program analysis where we want to infer properties of variables, and for program verification where it is desirable to have compact representations.

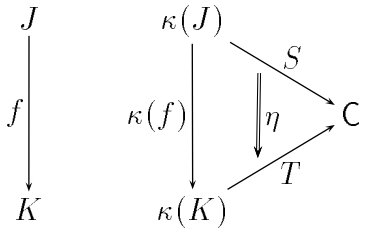
This paper constructs models of value-passing processes where, in contrast, the semantics is neither early nor late, where variable assignments are not expressed by branching, where loops are not unfolded and where there are no explicit fixpoints. Moreover, the resulting model is more in line with categorical models of other first-order theories particularly with respect to the interpretation of program variables. For example, we obtain message passing (substitution) by pullback similar to categorical models of predicate logic.

The paper rests on a generalization of labelled transition systems. Recall that a labelled transition system is essentially an edge labelled graph. More precisely, a transition system labelled in an alphabet L is a diagram in the category of graphs, Grph, of the form:

$$1 \xrightarrow{\triangledown} J \xrightarrow{f} G_L$$

Here the graph J is the *shape* of the transition system. The vertices of J are states and the edges are transitions. The graph G_L has a single vertex and an edge for each label in L. The morphism $\triangledown$ selects the start state.

We propose instead *categorical transition systems* (hereafter just *systems*). These generalize conventional transition systems in two ways. First we abandon the requirement that shapes be graphs. A universe of shapes is a category Shp such that each shape, $J \in \mathsf{Shp}$, determines a category $\kappa(J)$, via a functor: $\kappa : \mathsf{Shp} \longrightarrow \mathsf{Cat}$. Second, a shape is labelled in a category rather than an alphabet. Thus, a categorical transition system, $\mathbf{S}$, is a pair consisting of a



Diag. 1.

shape and a functor: $(J, \kappa(J) \xrightarrow{S} \mathsf{C})$. Systems form a category $\mathsf{CTS}(\mathsf{Shp}, \mathsf{C})$ where, given a second system $\mathbf{T} = (K, \kappa(K) \xrightarrow{T} \mathsf{C})$ a morphism $\mathbf{S} \longrightarrow \mathbf{T}$ is a pair (f, η) where $f : J \longrightarrow K$ is a morphism of shapes and $\eta : S \Longrightarrow T \circ \kappa(f)$ is a natural transformation (Diagram 1).

Abandoning graphs offers the opportunity of choosing shapes more suitable for modeling concurrent and asynchronous computation. Labelling a shape in a category means that both transitions and states can have more structure. For example, actions can be functions, machine instructions or even processes.

When $\mathsf{Shp} = \mathsf{Cat}$ and $\kappa = Id : \mathsf{Cat} \longrightarrow \mathsf{Cat}$, a category of systems reduces to a category of diagrams in C. Such categories are sometimes used to define limits and colimits as functors. Goguen has long advocated diagrams as semantic objects in computing. See [25,26] for an overview and further references. According to Goguen a diagram represents a *system* in a broad sense. The exact computational interpretation depends on the underlying category, but for example, the objects in the diagram may represent processes and the morphisms the interconnections. Diagrams can be constructed incrementally using colimits in the category of diagrams. The limit of a diagram represents the behaviour of the system.

By allowing shapes other than categories, categorical transition systems represent a modest generalization of diagrams. In fact, this paper does not exploit this generality and much of what follows fits within the paradigm advocated by Goguen. The novel contribution here is *twisted systems* which we describe shortly.

Like categories of diagrams, CTS arises as an instance of the Grothendieck construction as follows:

$$\mathsf{CTS}(\mathsf{Shp}, \mathsf{C}) = \int_{\mathsf{Shp}} \left(\mathsf{Shp}^{op} \xrightarrow{\kappa^{op}} \mathsf{Cat}^{op} \xrightarrow{\mathsf{C}^{(-)}} \mathsf{CAT} \right)$$

This means that results due to Tarlecki et al. [41] and Gray [29] can be used to infer when limits and colimits lift from the underlying categories to a category of systems. In particular, when Shp and C are complete, then the category of systems is also complete with limits constructed pointwise.

The remainder of the paper is structured as follows. The next section reviews an idea due to Burstall whereby flowcharts are presented as systems.

This provides intuition for later sections. Section 3 considers reflexive graphs as a universe of shapes. Then, in Section 4, we introduce a specific class of systems called twisted systems. Section 5 then outlines a general framework for typed processes using twisted systems. An instance of this is discussed in Section 6 where examples are provided to illustrate interaction and message passing by pullback. As an application of the theory, Section 7 introduces a simple parallel language which is given a categorical interpretation in Section 8. We conclude in Section 9.

2 Flowcharts as functors

We begin with an example of a category of systems which provides intuition for later sections. In [9] Rod Burstall describes how a flow-chart can be represented by a functor from a free category to the category of sets and partial functions, or more generally, sets and relations. A program is a system $(G, F(G) \xrightarrow{S} \mathsf{Rel})$ where G is a graph and $F(G)$ is the free category. Figure 1 illustrates this with the factorial program. The vertices of G are the states or *program points*. The vertex a is the start state and paths in G are computation paths. The image of each vertex is a cartesian product with a component for each variable in scope. Tuples in a product are the possible variable assignments at a particular program point. Transitions are labelled with relations. In the example we use terms from a typed λ-calculus extended with predicates to denote partial functions.

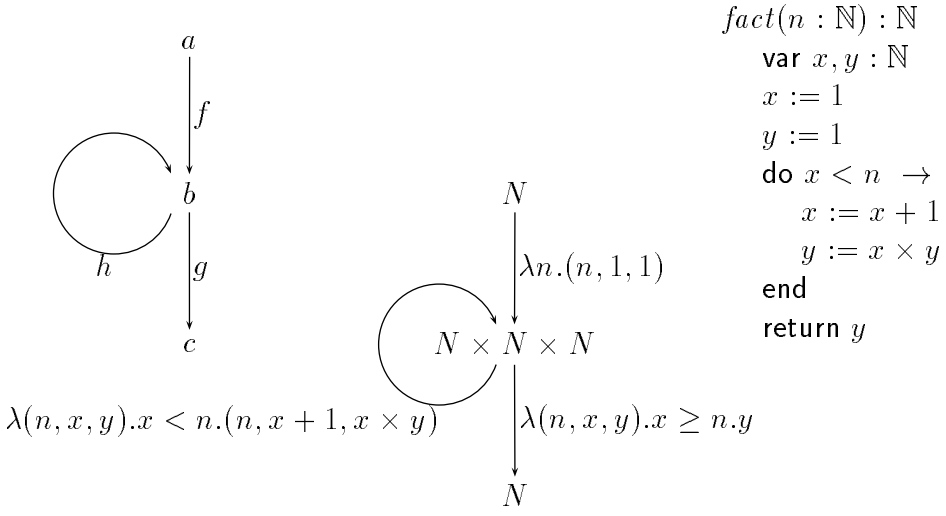
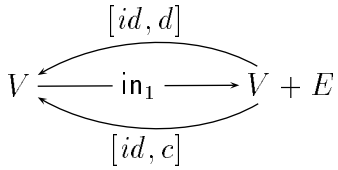


Fig. 1. Factorial $(G, F(G) \xrightarrow{S} \mathsf{Rel})$

Systems of this form differ from conventional flow-charts since, as shown in the example, the steps of the program are associated with the edges of the underlying graph rather than the vertices. Nevertheless, they express the same information and defining flow-charts as functors is simpler. Another difference



Diag. 2.

is that the actions in conventional flow-charts are program fragments and typically assignments or conditionals. Therefore, the correspondence between flow-charts and systems might be better if the target was a syntactic category rather than Rel. However, by choosing Rel, such a functor becomes not only an alternative representation of the program, but also a definition of its operational semantics. Operationally, each action in a system is a conditional rewrite rule which can "fire" only for those tuples in its preimage, or in other words, those tuples which satisfy the precondition for the rule.

3 Reflexive graphs

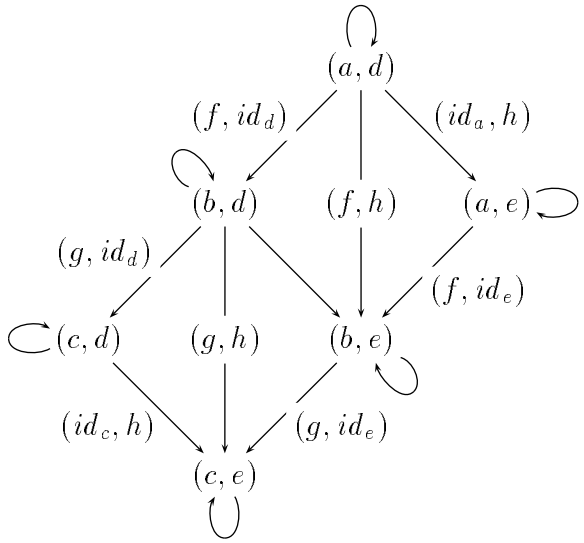
Next we turn our attention to shapes. There are many reasonable choices for universes of shapes including graphs and partial orders. A particularly interesting choice are higher-dimensional automata (hda) introduced by Pratt in which he adds higher-dimensional transitions to express concurrency [39]. This was expressed in terms of cubical sets by Goubault and Jensen [28,27]. In these terms an hda determines a category via a construction analogous to that for a fundamental groupoid for a topological space.

For the purposes of this paper, we will focus on the category of *reflexive graphs*, RGrph, as a universe of shapes. A reflexive graph is one in which every vertex has a designated "identity" edge. More precisely, a reflexive graph is a diagram in Set of the form shown in Diagram 2.

We interpret the identity edges in a reflexive graph as idling transitions. For the remainder of the paper the term "graph" will mean reflexive graph. Reflexive graphs as shapes for transition systems first appeared in [36]. See also [22].

The appearance of idling transitions has implications on limits. Diagram 3 illustrates the product of $a \xrightarrow{f} b \xrightarrow{g} c$ and $d \xrightarrow{h} e$. Like the product in Grph it contains the synchronous concurrent transitions (f, h) and (g, h). However it also has the interleaving of the transitions from the two processes.

In general, the product of a family of n shapes yields a collection of n-dimensional hypercubes. An m-dimensional transition where $m < n$ is one with m non-identity components. This represents the simultaneous occurrence of m actions from the component processes. Thus RGrph represents a simple model of concurrency in which *all* opportunities for concurrency arise from products and, moreover as we will see, where interaction arises from



Diag. 3.

limits. This differs from other models such as event structures, asynchronous transition systems and higher-dimensional transition systems which come equipped with a mechanism for expressing opportunities for concurrency but where that mechanism is inconsistent with the opportunities which arise from forming products [43,11].

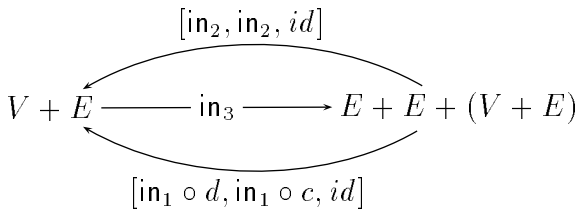
Conventional transition systems have a start state. By analogy a *pointed graph* is a pair $(J, \triangledown_J : 1 \longrightarrow J)$ for some graph J. The category of pointed graphs is the comma category ${}^{\triangledown}\mathsf{RGrph} = 1 \downarrow \mathsf{RGrph}$. The required functor to Cat simply forgets the point and forms the free category. The language we define later has sequential composition (rather than prefixing) for which we require shapes having some number of end states. We limit ourselves to one and define the category of *bipointed graphs* as ${}^{\triangledown}_{\triangle}\mathsf{RGrph} = 1 + 1 \downarrow \mathsf{RGrph}$. Typically we will simply omit mentioning the start and end states. We will write simply J for a shape and refer to $\triangledown_J$ and $\triangle_J$ when needed. A morphism $J \longrightarrow K$ is a graph homomorphism which preserves the start and end states.

4 Twisted systems

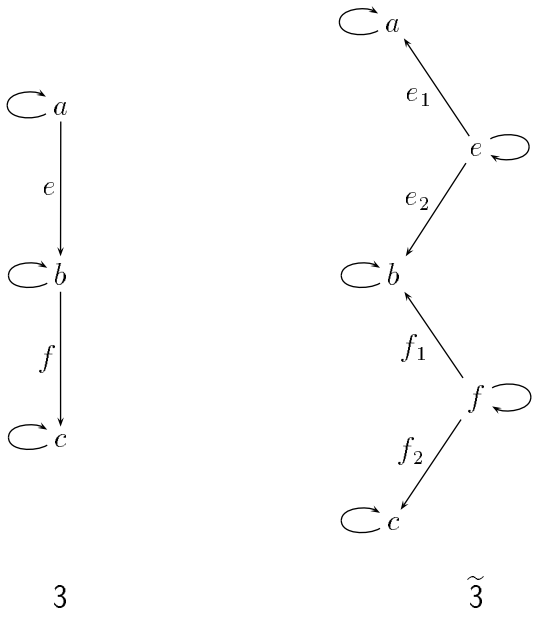
The class of systems relevant to this paper are *twisted systems*. These have the form:

$$\mathbf{S} = (J, F\widetilde{J} \longrightarrow \mathsf{C})$$

where J is a bipointed reflexive graph and $(\widetilde{-})$ is the *twisted graph* construction. This is a graph formed by replacing every edge $a \xrightarrow{e} b$ by a span $a \longleftarrow e \longrightarrow b$. To make this precise, given a graph as defined in Diagram 2, the twisted graph has the form shown in Diagram 4.



Diag. 4.



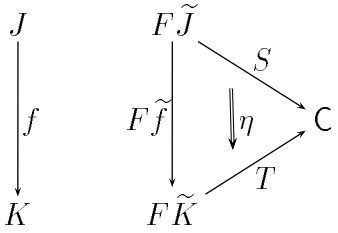
Diag. 5.

An example of a graph and its "twist" is shown in Diagram 5. It is not difficult to show that the twist construction is functorial $(\widetilde{-}) : \mathsf{RGrph} \longrightarrow \mathsf{RGrph}$. Note also that for any graph J, constructing the free category $F\widetilde{J}$ adds no arrows whatsoever.

This is an instance of a more general class of twisted systems described in [18]. The more general class uses the *twisted arrow category* and hence the use of the term "twist" here.

Our interest in this construction lies in the relationship between twists and spans. Let $\mathsf{Sp}(\mathsf{C})$ be the bicategory of spans over a category C with pullbacks. Each edge $e : a \longrightarrow b$ in J determines a span $a \xleftarrow{e_1} e \xrightarrow{e_2} b$ in $\widetilde{J}$. In [18,20] it is shown that a functor $S : FJ \longrightarrow \mathsf{Sp}(\mathsf{C})$ is essentially the same as a functor $S' : F\widetilde{J} \longrightarrow \mathsf{C}$ insofar as they select equivalent objects and arrows in C. Spans generalize both partial functions and relations. This means that rather than presenting flow-charts as functors $FJ \longrightarrow \mathsf{Rel}$ we can use presheaves, $F\widetilde{J} \longrightarrow \mathsf{Set}$.

The categories $\mathsf{C}^{F\widetilde{J}}$ and $\mathsf{Sp}(\mathsf{C})^{FJ}$ differ significantly in their morphisms. A natural transformation in $\mathsf{C}^{F\widetilde{J}}$ is, in effect, a collection of arrows in C indexed



Diag. 6.

by the edges of J rather than a collection of spans indexed by the vertices of J. In fact natural transformations in $\mathsf{C}^{F\widetilde{J}}$ correspond to a specific class of *oplax natural transformations* between functors in $\mathsf{Sp}(\mathsf{C})^{FJ}$. A more precise statement of this relationship is given in the appendix.

Twisted systems form a category $\mathsf{TwS}(\mathsf{C})$ where, as before, an object is a pair $(J, F\widetilde{J} \xrightarrow{S} \mathsf{C})$ in which the domain of S is twisted. Given objects $\mathbf{S} = (J, F\widetilde{J} \xrightarrow{S} \mathsf{C})$ and $\mathbf{T} = (K, F\widetilde{K} \xrightarrow{T} \mathsf{C})$, a morphism $\mathbf{f} : \mathbf{S} \longrightarrow \mathbf{T}$ is a pair (f, η) where $f : J \longrightarrow K$ is a graph homomorphism and $\eta : S \Longrightarrow T \circ F\widetilde{f}$ is a natural transformation (Diagram 6).

Our intuition is that S associates a span in C with every transition in J. Note that if $\mathsf{C} = \mathsf{Set}$ and $e : a \longrightarrow b$ is an edge in J such that the apex $Se = 0$, then following the operational reading given earlier for functors into Rel, such a transition can never happen or "fire". We call these *null transitions*. It follows that in twisted systems there are two notions of deadlock. A state a is said to exhibit *path deadlock* when the only transition leaving it is the identity. The state exhibits *data deadlock* when all the non-identity transitions leaving a are null transitions.

5 Typed processes

Let $\mathbf{U}$ be a system. We define a process of type $\mathbf{U}$ to be a system $\mathbf{P}$ and a morphism $\mathbf{p} : \mathbf{P} \longrightarrow \mathbf{U}$ in $\mathsf{TwS}(\mathsf{Set})$. The family of all processes of type $\mathbf{U}$ is thus the category $\mathsf{TwS}(\mathsf{Set}) \downarrow \mathbf{U}$.

The object $\mathbf{U}$ represents the *interface* to the process. It defines what can be observed and what the process offers to the environment and to its correspondents for interaction. The system $\mathbf{P}$ describes the internal behaviour of the process and the morphism $\mathbf{p}$ expresses how the internal system realizes observable behaviour. Typically an interface will render some aspects of the internal system unobservable.

An interface is typically a product $\mathbf{U} = \mathbf{V}_1 \times \cdots \times \mathbf{V}_n$ where each component represents a communication channel. Processes can interact when they share a channel or channels. In the simplest communication mode, if $\mathbf{p} : \mathbf{P} \longrightarrow \mathbf{U} \times \mathbf{V}$ and $\mathbf{q} : \mathbf{Q} \longrightarrow \mathbf{V} \times \mathbf{W}$ are processes, then they compose by pullback over the common channel $\mathbf{V}$ to yield the process $\mathbf{p} \parallel_{\mathbf{V}} \mathbf{q} : \mathbf{R} \longrightarrow \mathbf{U} \times \mathbf{W}$.

Using the associativity and commutativity of products, the channels in an interface can ordered and grouped as necessary.

As an example, suppose L is an alphabet and we work in the category $\mathsf{TwS}(L^{\circledast})$ where $L^{\circledast}$ is the free algebraic theory for L. Composition by pullback then yields communication behaviour similar to both Milner's CCS and Hoare's CSP insofar as interaction requires simultaneous participation on the part of both correspondents. It differs from both CCS and CSP in that channels are shared by exactly two processes. After composition, the channel is hidden.

An interface and its channels encode a protocol which must be followed by a pair of correspondents. These protocols have both spatial and temporal dimensions. By spatial we refer to the channels in a product which may serve different roles. By temporal we refer to the fact that interface is itself a system with transitions which evolves.

The separation of internal and observable behaviour as described above is not new. It appears in Ferrari et al. [21,22] and has been advocated by Goguen [25,26] in his theory of systems. As here, Goguen defines the behaviour of a system to be the limit of a diagram. Another instance is due to Cockett and Spooner [14–16]. They construct categories in which morphisms are typed processes. Processes are spans in a category of conventional transition systems. As here, composition is by pullback.

Whether a process is a span as in Cockett and Spooner or an object in a comma category as here is not important. They express precisely the same information. Defining processes as spans is an appealing option. Doing so yields an *interaction category* in the sense of Abramsky [2,3,1] in which processes have linear types [24]. Also, it is not difficult to show that (the classifying category for) a bicategory of spans is a *traced monoidal category* in the sense of Joyal et al. [33]. In spite of this, it is usually simpler to have a single interface and use projections to isolate channels as necessary.

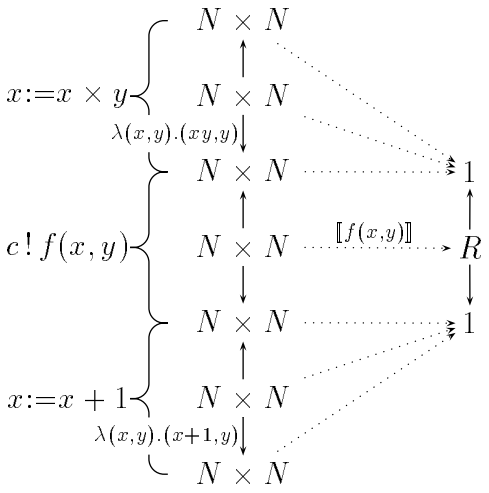
6 Value-passing by pullback

Before we formally define a parallel language and give its semantics, we will illustrate how pullbacks model parallel composition and value passing. We work in the category $\mathsf{TwS}(\mathsf{Set})$.

Consider the following simple program written in a simple parallel language with CSP-style input/output primitives.

$$\mathcal{P} = \begin{cases} x := x \times y \\ c\,!\,f(x,y) \\ x := x + 1 \quad [x\!:\!\mathbb{N}, y\!:\!\mathbb{N}, f\!:\!\mathbb{N}\times\mathbb{N}\longrightarrow\mathbb{R}] \end{cases}$$

The process transmits data on the channel c where the channel has at least



Diag. 7.

one transition carrying data of type real.

The process is interpreted by a morphism $\mathbf{p} : \mathbf{P} \longrightarrow \mathbf{U}$ shown in Diagram 7. The column of spans on the left depict the system $\mathbf{P}$. This captures the internal behaviour of the process. Idling transitions have been omitted and the unlabelled morphisms are identities.

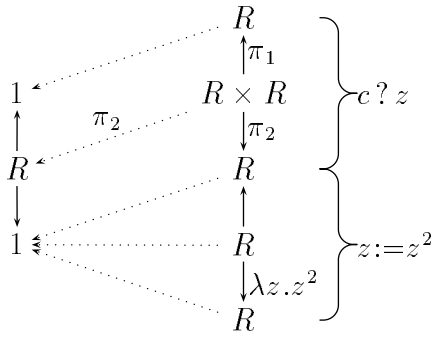
The single span on the right is the system for the interface $\mathbf{U}$ representing the channel c. The dashed lines between the two systems depicts the morphism between them. They show both the graph homomorphism relating the shapes and the components of the natural transformation. All the unlabelled components are terminal arrows and the single labelled morphism represents the output to the channel.

As discussed above, the interface $\mathbf{U}$ describes what is observable by the environment. Note the two occurrences of the terminal object. These imply that, unlike $\mathbf{P}$, the interface has no variables nor memory. However, the terminal objects in the interface have a more subtle effect. First not only does the interface have no internal state, but the internal state of $\mathbf{P}$ is rendered unobservable. Also note that the first and last transitions of $\mathbf{P}$ are mapped to id_1. Consequently neither transition interacts with the environment and neither transition is observable. These are analogous to non-preemptive τ transitions in CCS.

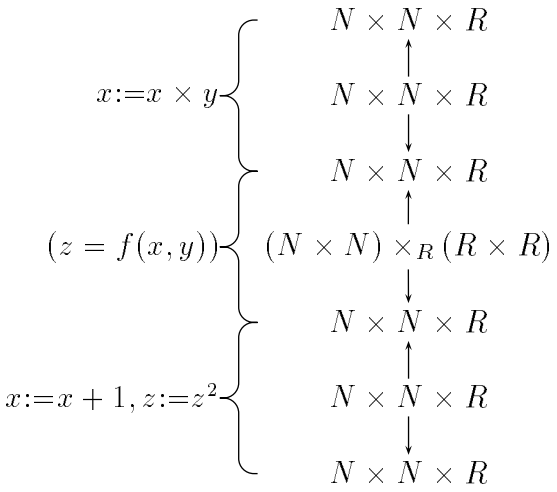
Now consider a fragment of a second process $\mathbf{q} : \mathbf{Q} \longrightarrow \mathbf{U}$ shown in Diagram 8 and having the same interface as $\mathbf{p}$.

$$\mathcal{Q} = \begin{cases} c\,?\,z \\ z := z^2 \quad [z : \mathbb{R}] \end{cases}$$

Note in particular the occurrence of the object $R \times R$ and the use of projection maps. The second R represents a new instance of the variable



Diag. 8.



Diag. 9.

z which is universally quantified. Any constraints on z from earlier in the program are discarded. The value of z used in the subsequent assignment is the value bound by the input command.

The parallel composition of the two processes is the pullback over $\mathbf{U}$. This is shown in Diagram 9. The objects in the image of the functor for the system are obtained by the respective pullbacks of the components of the natural transformations in $\mathbf{p}$ and $\mathbf{q}$. All the morphisms in the system are universal.

There are a few observations to be made of the composed process. First, it illustrates substitution by pullback as for example in categorical models of predicate logic. Second, pulling back components of natural transformations over the terminal object yields a product. Consequently, as might be expected, the type of the internal state of the composed process is the product of the types of the individual processes.

A number of issues remain. The first is that, while it may be clear that the system formed by the pullback above faithfully records the constraints on variables which arise from interaction, those constraints are not propagated forward. For example, composing the following two processes yields a system

with two (non-idling) transitions, however, the post-condition of the second transition does not reflect the fact that z is even.

$$\mathcal{P} = \begin{cases} c\,?\,x & \\ z := x + 2 \quad [x:\mathbb{N}, z:\mathbb{N}] & \end{cases} \qquad \mathcal{Q} = c\,!\,2y \quad [y:\mathbb{R}]$$

The reader is referred to [18] for the definition of a coreflective subcategory of *forward systems* which incorporates such propagation. The example above and those which follow are easily adapted to this subcategory.

The next issue is to decide on an interpretation for the composition of the following:

$$\mathcal{P} = c\,?\,x \quad [x:\mathbb{N}] \qquad \mathcal{Q} = c\,?\,y \quad [y:\mathbb{N}]$$

In practice it is reasonable to allow two or more processes to listen on the same channel. A message sent to c might be broadcast to all receivers or delivered to one chosen non-deterministically. Similarly we might wish to allow many processes to send on the same channel. However, this paper will consider only simple one-to-one communication in which each channel is shared by exactly two processes.

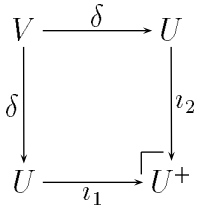
Under this assumption, there are two options with respect to composing $\mathcal{P}$ and $\mathcal{Q}$. We can either deem $\mathcal{P} \parallel \mathcal{Q}$ to be ill-formed or, if we accept the program, to give it a semantics which reflects the deadlock. Achieving the former might be done with a suitable type discipline. This is perhaps the preferred solution but will not be considered here.

The solution adopted here is to associate a direction of flow with each message through a channel. This information can be coded in different ways. Two are described below.

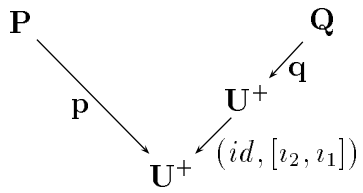
The first is to encode the direction as part of the message. If a channel is to carry data of sort $\mathbb{X}$ in a transition $a \xrightarrow{e} b$, then the interface labels the apex e with the set $[\![\mathbb{X}]\!] + [\![\mathbb{X}]\!]$ with a protocol where, for example, input is in the left component and output is in the right. To make this precise, assume $\mathbf{U} = (L, U)$ is the system for a channel c without direction information. We wish to define a new system similar to $\mathbf{U}$ in which every transition is labelled by a coproduct and where every state is still labelled by the terminal. To do so, define the functor $V : F\widetilde{L} \longrightarrow \mathsf{Set}$ such that for each state $a \in L$, $Va = 1$ and for all edges $e : a \longrightarrow b$ in L, $Ve = 0$. This gives a universal natural transformation $\delta : V \Longrightarrow U$. Now construct the system $\mathbf{U}^+ = (L, U^+)$ where U^+ is the functor obtained by the pushout in Diagram 10.

Processes are now defined relative to $\mathbf{U}^+$. The input/output roles must be reversed in one correspondent for there to be communication. For this purpose we define the channel automorphism:

$$(id, [\imath_2, \imath_1]) : \mathbf{U}^+ \longrightarrow \mathbf{U}^+$$



Diag. 10.



Diag. 11.

This serves as a *null modem*. Composition is by pullback as in Diagram 11. For the motivating example above this yields a system with a single null transition.

A second way to distinguish input from output is in the shapes. Non-identity edges in the graph for the interface are duplicated with one edge in each pair for input and the other for output. Doubling edges in a shape is achieved by the pushout in Diagram 12 where $|K|$ is the discrete graph formed from the vertices of K and $\Phi_K : |K| \longrightarrow K$ is the obvious embedding. This extends to a *transition doubling* endofunctor on shapes:

$$(-)^{db} : {}^{\triangledown}_{\triangle}\mathsf{RGrph} \longrightarrow {}^{\triangledown}_{\triangle}\mathsf{RGrph}$$

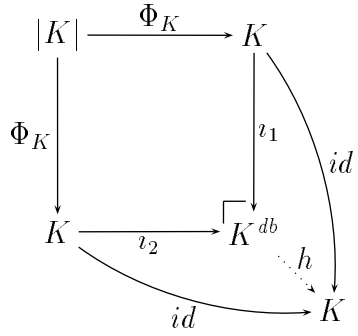
and a corresponding functor on systems such that:

$$\mathbf{U}^{db} = (K^{db}, F\widetilde{K^{db}} \xrightarrow{F\tilde{h}} F\widetilde{K} \xrightarrow{U} \mathsf{Set})$$

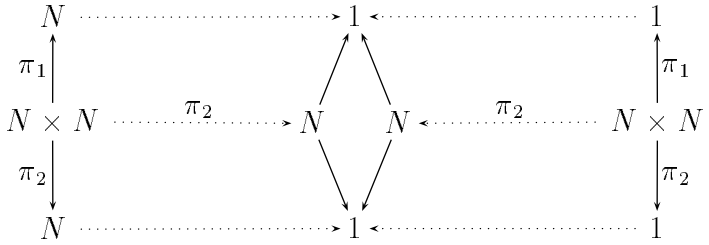
In this case the null modem is the automorphism:

$$([\iota_2, \iota_1], id) : \mathbf{U}^{db} \longrightarrow \mathbf{U}^{db}$$

Diagram 13 applies the construction to the example above. On the left and right are the internal systems for the processes $\mathcal{P}$ and $\mathcal{Q}$ with the shared interface in the middle. The latter has shape 2^{db}. If we adopt the protocol that the left branch of 2^{db} is for input, then in the diagram, the null modem has been incorporated into the morphism on the right. The process $\mathcal{P} \parallel \mathcal{Q}$ has no transitions. So whereas encoding the direction of flow as part of the message leads to data deadlock, encoding the direction in the shape yields path deadlock.



Diag. 12.



Diag. 13.

7 A simple parallel language

In this section we introduce a simple imperative parallel language similar to Occam [35]. The next section gives a categorical semantics. The language is an extension of the sequential language studied in [20].

It is convenient to assume that the basic types and expressions in the language are provided by an algebraic theory, Th. This will be left unspecified but it is assumed to contain types and standard constants for the natural numbers $\mathbb{N}$, and booleans, $\mathbb{B}$. The collection of well-formed terms in the algebra are generated in the usual way by rules containing judgements or *terms in context* of the form $t : \mathbb{X}\ [\Gamma]$. Here t is a term, $\mathbb{X}$ a sort and $\Gamma = [x_1 : \mathbb{X}_1, \ldots, x_n : \mathbb{X}_n]$ is a *context* containing a list of typed variables including the free variables in t. The algebraic theory is equipped with a collection of axioms and rules formed from judgements or *equations in context* written $t_1 =_{Th} t_2\ [\Gamma]$. The theory determines a syntactic or *classifying* category $\mathsf{Cl}(Th)$. It is assumed there is semantic finite product preserving functor $[\![-]\!] : \mathsf{Cl}(Th) \longrightarrow \mathsf{Set}$ such that:

$$[\![t : \mathbb{X}\ [\Gamma]]\!] = [\![\Gamma]\!] \xrightarrow{[\![t]\!]} [\![\mathbb{X}]\!]$$

where $[\![\Gamma]\!] = [\![\mathbb{X}_1]\!] \times \cdots \times [\![\mathbb{X}_n]\!]$. See Crole [17] or Pitts [38] for an introduction to categorical models of algebraic theories.

The syntax of the programming language is defined on top of the algebra.

The grammar is as follows:

$$
\begin{aligned}
gc &::= \mathsf{nil} \mid act \mid \mathsf{var}\ x : \mathbb{X}\ \mathsf{in}\ gc \\
&\quad \mid gc\ ;\ gc \mid \mathsf{if}\ alt \mid \mathsf{do}\ alt \\
&\quad \mid \mathsf{chan}\ c_1 : \mathbb{X}_1, \ldots, c_n : \mathbb{X}_n\ \mathsf{in}\ gc \parallel gc \\
alt &::= b \to gc \,[]\ldots[]\, b \to gc \\
act &::= x := t \mid c\,!\,t \mid c\,?\,x
\end{aligned}
$$

Here, b is a predicate on the state, x is a typed variable and t is a term. All come from the underlying algebra. Often we will write $[]_{i=1}^{n}(b_i \to gc_i)$ in place of $b_1 \to gc_1 \,[]\cdots[]\, b_n \to gc_n$.

A *program in context* is a judgement of the form $gc\ [\Gamma|\Theta]$. Here Γ is *variable context* containing a list of typed variables currently in scope. Similarly Θ is a *channel context* listing the channels currently in scope. The rules for well-formed programs appear in Figure 2. Note that the two parts of a context implies that a process is typed in two ways. The variable context reflects the type for sequential composition and the channel context reflects the type for parallel composition.

The informal meaning of conditionals and repetitions are as follows. Given a variable assignment s, the if is evaluated by choosing non-deterministically one of the guarded commands from those whose guard is satisfied by s. If none of the guards are satisfied the program deadlocks. In the case of do, the selection of a guarded command is repeated until no guard is satisfied whereupon the loop terminates successfully.

Processes communicate through typed channels. Channel types are simple and similar to Occam [35]. A declaration $\mathsf{chan}\ c : \mathbb{X}$ means that c carries values of type $\mathbb{X}$ where $\mathbb{X}$ is a sort from the underlying algebra. Processes can communicate any number of times across the channel and in either direction but only data of the specified type may be carried.

The intended meaning of $\mathsf{chan}\ c_1 : \mathbb{X}_1, \ldots, c_n : \mathbb{X}_n\ \mathsf{in}\ gc_1 \parallel_c gc_2$ is that first the channels $c_1, \ldots, c_n$ are allocated but visible only to gc_1 and gc_2. They proceed in parallel exchanging messages through the specified channels. The processes synchronize again at the end of the block where the channels are discarded.

The side condition on the rule for parallel composition forces the variable and channel contexts of the two processes to be disjoint. This ensures that the processes do not interfere with one another and that the only contact is through the specified channels. The remaining rules are straightforward.

$$\frac{}{\mathsf{nil}\ [\Gamma \mid \Theta]} \qquad \frac{t : \mathbb{X}\ [\Gamma, x : \mathbb{X}]}{x := t\ [\Gamma, x : \mathbb{X} \mid \Theta]}$$

$$\frac{(b_i : \mathbb{B}\ [\Gamma])_{i=1}^n \qquad (gc_i\ [\Gamma \mid \Theta])_{i=1}^n}{\mathsf{if}\ [\!]_{i=1}^n (b_i \to gc_i)\ [\Gamma \mid \Theta]} \qquad \frac{(b_i : \mathbb{B}\ [\Gamma])_{i=1}^n \qquad (b_i \to gc_i\ [\Gamma \mid \Theta])_{i=1}^n}{\mathsf{do}\ [\!]_{i=1}^n (b_i \to gc_i)\ [\Gamma \mid \Theta]}$$

$$\frac{gc\ [\Gamma, x : \mathbb{X} \mid \Theta]}{\mathsf{var}\ x : \mathbb{X}\ \mathsf{in}\ gc\ [\Gamma \mid \Theta]}\ x \notin \Gamma \qquad \frac{gc_1\ [\Gamma \mid \Theta] \qquad gc_2\ [\Gamma \mid \Theta]}{gc_1\ ;\ gc_2\ [\Gamma \mid \Theta]}$$

$$\frac{gc\ [\Gamma \mid \Theta]}{gc\ [\Gamma, x : \mathbb{X} \mid \Theta]}\ x \notin \Gamma \qquad \frac{gc\ [\Gamma \mid \Theta]}{gc\ [\Gamma \mid \Theta, c : \mathbb{X}]}\ c \notin \Theta$$

$$\frac{gc\ [\Gamma_1, \Gamma_2 \mid \Theta]}{gc\ [\Gamma_2, \Gamma_1 \mid \Theta]} \qquad \frac{gc\ [\Gamma \mid \Theta_1, \Theta_2]}{gc\ [\Gamma \mid \Theta_2, \Theta_1]}$$

$$\frac{gc_1\ [\Gamma_1 \mid \Theta_1, \Theta] \qquad gc_2\ [\Gamma_2 \mid \Theta_2, \Theta]}{\mathsf{chan}\ \Theta\ \mathsf{in}\ gc_1 \parallel gc_2\ [\Gamma_1, \Gamma_2 \mid \Theta_1, \Theta_2]}\ \Gamma_1 \cap \Gamma_2 = \emptyset, \Theta_1 \cap \Theta_2 = \emptyset$$

$$\frac{t : \mathbb{X}\ [\Gamma]}{c\,!\,t\ [\Gamma \mid \Theta, c : \mathbb{X}]} \qquad \frac{}{c\,?\,x\ [\Gamma, x : \mathbb{X} \mid \Theta, c : \mathbb{X}]}$$

Fig. 2. Well-formed programs in context.

8 Categorical semantics

The remainder of the paper gives an interpretation of the parallel language in the category $\mathsf{TwS}(\mathsf{Set})$. The meaning of program in context $gc\ [\Gamma \mid \Theta]$ will be a process $\mathbf{p} : \mathbf{P} \longrightarrow \mathbf{U}$ where $\mathbf{U}$ is the interface interpreting Θ and $\mathbf{P}$ is system representing the algorithm gc. Thus, a complete specification requires two systems and a morphism.

It is important to bear in mind that there are two type disciplines here. Processes have separate types for sequential and parallel composition.

We begin by discussing the interpretation of channels and channel contexts. Let $\mathbb{X}$ be a sort in Th. Since channels carry a single data type perpetually, we define the system $\mathbf{X} = (M, X)$ where M is the graph having a single non-idling transition $k \circlearrowright \bullet$ in which $\bullet$ is both the start and end state. The functor

$X : F\widetilde{M} \longrightarrow \mathsf{Set}$ is given by:

$$X\bullet = 1$$
$$Xk = [\![\mathbb{X}]\!]$$

With $\nabla = \triangle$ we have a third way of indicating the direction of flow through a channel. A channel $c : \mathbb{X}$ is interpreted by the product $\mathbf{X} \times \mathbf{X}$ with the protocol that the left port is for input and the right is output. As before, these roles must be reversed in one correspondent for there to be communication. For this we define the channel automorphism (null modem):

$$\alpha_X = \langle \pi_2, \pi_1 \rangle : \mathbf{X} \times \mathbf{X} \longrightarrow \mathbf{X} \times \mathbf{X}$$

If $\Theta = [c_1 : \mathbb{X}_1, \ldots, c_n : \mathbb{X}_n]$ is a channel context, then Θ is interpreted by interface formed from the product of its channels:

$$\mathbf{U} = (\mathbf{X}_1 \times \mathbf{X}_1) \times \cdots \times (\mathbf{X}_n \times \mathbf{X}_n)$$

We extend the definition of the null modem to contexts and define:

$$\alpha_\Theta = \langle \pi_2, \pi_1 \rangle \times \cdots \times \langle \pi_2, \pi_1 \rangle : \mathbf{U} \longrightarrow \mathbf{U}$$

Often it is necessary to define processes which have no observable behaviour on any channel in a context. Given an internal system $\mathbf{P} = (J, P)$ and an interface $\mathbf{U}$, define the TwS morphism $\tau_{\mathbf{P},\mathbf{U}} = (f, \eta) : \mathbf{P} \longrightarrow \mathbf{U}$ such that for all edges $e \in J$:

$$f(e) = id\bullet$$
$$\eta_e = ! : Pe \longrightarrow 1$$

The morphism $\tau_{\mathbf{P},\mathbf{U}}$ renders all of the transitions in $\mathbf{P}$ unobservable with respect to the interface $\mathbf{U}$ as illustrated in the example given earlier. The name refers to τ transitions in CCS as unobservable actions.

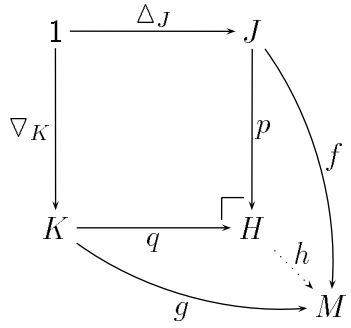
We now turn to the interpretation of programs in context $gc\ [\Gamma \mid \Theta]$. We assume the channel context Θ is interpreted by the system $\mathbf{U} = (L, U)$ as discussed above. It remains to define the internal system $\mathbf{P}$ and the morphism $\mathbf{P} \longrightarrow \mathbf{U}$ for each phrase type.

Empty program

The system $\mathbf{P}$ for the program $\mathsf{nil}\ [\Gamma \mid \Theta]$ is $(1, F\widetilde{1} \xrightarrow{[\![\Gamma]\!]} \mathsf{Set})$. It has no observable behaviour hence the morphism to the interface is $\tau_{\mathbf{P},\mathbf{U}}$.

Sequential composition

We are given a pair of program fragments and their interpretations as summarized in the following table.



Diag. 14.

fragment	$gc_1\ [\Gamma \mid \Theta]$	$gc_2\ [\Gamma \mid \Theta]$
internal system	$\mathbf{P} = (J, P)$	$\mathbf{Q} = (K, Q)$
morphism to interface	$(f, \theta) : \mathbf{P} \longrightarrow \mathbf{U}$	$(g, \phi) : \mathbf{Q} \longrightarrow \mathbf{U}$

Assume also that $\mathbf{U} = (M, U)$. Note that, according to the rules for well-formed programs, the variable contexts of the two processes must be the same. Thus $P\widetilde{\triangle}_J = Q\widetilde{\nabla}_K$. Our goal is to append a copy of $\mathbf{Q}$ to the end point of $\mathbf{P}$ and construct a morphism to the common interface. This will yield a system $\mathbf{R} = (H, R)$ and the morphism $(h, \epsilon) : \mathbf{R} \longrightarrow \mathbf{U}$.

The shape H is obtained from the pushout in Diagram 14. This identifies the start state of K with the end state of J. The diagram also shows the universal map $h : H \longrightarrow M$.

For the functor $R : F\widetilde{H} \longrightarrow \mathsf{Set}$, we first twist the span from Diagram 14 and form a second pushout as shown in Diagram 15. It is easy to see that, while the second pushout is constructed in Cat, no free arrows are introduced in L and that L is isomorphic to $F\widetilde{H}$. Moreover, since P and Q agree at the vertices being identified, there is a universal map $D : \mathsf{L} \longrightarrow \mathsf{Set}$. The functor R is given by:

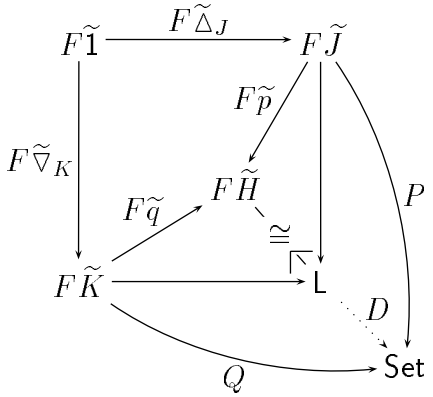
$$R = F\widetilde{H} \xrightarrow{\cong} \mathsf{L} \xrightarrow{D} \mathsf{Set}$$

The natural transformation $\epsilon : R \Longrightarrow U \circ F\widetilde{h}$ can be constructed as a universal arrow or defined directly. Bearing in mind that $\theta_{\triangle_J}$ and ϕ_{∇_K} are both arrows to the terminal, then $\epsilon_e = \theta_e$ when e is an edge from J and $\epsilon_e = \phi_e$ when e is from K.

Assignment

Let $\mathsf{2}$ be the graph $\nabla \longrightarrow \triangle$. The internal system $\mathbf{P}$ for the phrase $x := t\ [\Gamma_1, x : \mathbb{X}, \Gamma_2]$ is $(\mathsf{2}, P)$ where P labels the transition with the span

$$[\![\Gamma_1]\!] \times [\![\mathbb{X}]\!] \times [\![\Gamma_2]\!] \xleftarrow{id} [\![\Gamma_1]\!] \times [\![\mathbb{X}]\!] \times [\![\Gamma_2]\!] \xrightarrow{\langle \pi_{\Gamma_1}, [\![t]\!], \pi_{\Gamma_2} \rangle} [\![\Gamma_1]\!] \times [\![\mathbb{X}]\!] \times [\![\Gamma_2]\!]$$



Diag. 15.

As there is no interaction with the environment, the morphism to the interface is $\tau_{\mathbf{P},\mathbf{U}}$.

Local variables

Let $\mathbf{p} : \mathbf{P} \longrightarrow \mathbf{U}$ be the interpretation of gc in $\mathsf{var}\ x : \mathbb{X}\ \mathsf{in}\ gc\ [\Gamma \mid \Theta]$. To introduce a new variable, $\mathbf{p}$ is prefixed by a simple program $\mathbf{a}$ in which the variable is allocated and suffixed by the reciprocal program, $\mathbf{a}^\circ$, where the variable is discarded. The system which allocates the variable is $\mathbf{A} = (2, A)$ with a single transition labelled by A with the span $[\![\Gamma]\!] \xleftarrow{\pi} [\![\Gamma, x : \mathbb{X}]\!] \xrightarrow{id} [\![\Gamma, x : \mathbb{X}]\!]$. The transition in $\mathbf{A}$ is not observable, so define $\mathbf{a} = \tau_{\mathbf{A},\mathbf{U}} : \mathbf{A} \longrightarrow \mathbf{U}$. The final process is the sequential composition of $\mathbf{a}$ followed by $\mathbf{p}$ followed by $\mathbf{a}^\circ$. Note that variables are universally quantified (uninitialized) when introduced.

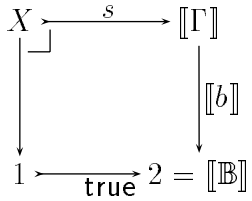
Input

Consider the program in context $c\,?\,x\ [\Gamma, x : \mathbb{X} \mid \Theta, c : \mathbb{X}]$. We construct $\mathbf{P} = (2, P)$ where P labels the span with:

$$[\![\Gamma]\!] \times [\![\mathbb{X}]\!] \xleftarrow{\langle \pi_1, \pi_2 \rangle} [\![\Gamma]\!] \times [\![\mathbb{X}]\!] \times [\![\mathbb{X}]\!] \xrightarrow{\langle \pi_1, \pi_3 \rangle} [\![\Gamma]\!] \times [\![\mathbb{X}]\!]$$

Here, the extra term in the product at the apex and the choice of projections reflects the fact that receiving input discards any previous constraints on the variable. Let $\mathbf{U}$ and $\mathbf{X} \times \mathbf{X}$ be the systems interpreting Θ and c respectively. Define the morphism $(f, \eta) : \mathbf{P} \longrightarrow \mathbf{X}$ where the morphism f is defined by $f(\nabla \xrightarrow{e} \triangle) = k : \bullet \longrightarrow \bullet$ and $\eta_e = \pi_3$. Bearing in mind that the left port is for input, we obtain a morphism $\langle (f, \eta), \tau_{\mathbf{P},\mathbf{X}} \rangle : \mathbf{P} \longrightarrow \mathbf{X} \times \mathbf{X}$. The action has no effect on any other channels so we construct:

$$\mathbf{p} = \langle \tau_{\mathbf{P},\mathbf{U}}, \langle (f, \eta), \tau_{\mathbf{P},\mathbf{X}} \rangle \rangle : \mathbf{P} \longrightarrow \mathbf{U} \times (\mathbf{X} \times \mathbf{X})$$



Diag. 16.

Output

The internal system $\mathbf{P}$ for the process interpreting $c!t\ [\Gamma \mid \Theta, c:\mathbb{X}]$ has a single transition labelled $[\![\Gamma]\!] \xleftarrow{id} [\![\Gamma]\!] \xrightarrow{id} [\![\Gamma]\!]$. For the morphism to the interface we proceed as with input. Define $(f,\eta) : \mathbf{P} \longrightarrow \mathbf{X}$ where f is as above and $\eta_e = [\![t]\!] : [\![\Gamma]\!] \longrightarrow [\![\mathbb{X}]\!]$. Output happens in the right port. Define:

$$\mathbf{p} = \langle \tau_{\mathbf{P},\mathbf{U}}, \langle \tau_{\mathbf{P},\mathbf{X}}, (f,\eta)\rangle\rangle : \mathbf{P} \longrightarrow \mathbf{U} \times (\mathbf{X} \times \mathbf{X})$$

Alternation

Before looking at conditionals and loops we consider alternation. Let $\mathbf{p}$ be the interpretation of gc in the single guarded command $b \to gc\ [\Gamma \mid \Theta]$. A predicate $b : \mathbb{B}\ [\Gamma]$ determines a subobject of $[\![\Gamma]\!]$ by pulling back against the classifying arrow in Set (Diagram 16).

Now construct $\mathbf{g} = \tau_{\mathbf{G},\mathbf{U}} : \mathbf{G} \longrightarrow \mathbf{U}$ where $\mathbf{G}$ has a single transition labelled by the span

$$[\![\Gamma]\!] \xleftarrow{s} X \xrightarrow{s} [\![\Gamma]\!]$$

This is a transition which can be taken only for tuples satisfying the predicate $[\![b]\!]$. The meaning of the single guarded command is the sequential composition of $\mathbf{g}$ followed by $\mathbf{p}$.

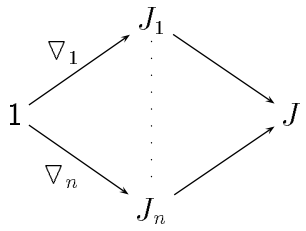
Now consider the family of guarded commands $(b_i \to gc_i\ [\Gamma \mid \Theta])_{i=1}^n$ where the meaning of each member is given by $\mathbf{P}_i = (J_i, P_i)$ and a morphism (f_i, η_i). The meaning of the alternation is the system $\mathbf{P} = (J, P)$ and the morphism $(f, \epsilon) : \mathbf{P} \longrightarrow \mathbf{U}$

The shape J is the colimit on the left of Diagram 17. This identifies respectively all the start states $(\triangledown_i)_{i=1}^n$. The construction of P, f, and ϵ then follows the same pattern as for sequential composition.

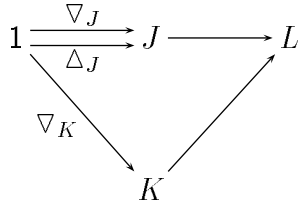
This scheme is easily extended to accommodate input and output in guards as found in Occam and Ada.

Conditional

The meaning of a conditional $\mathsf{if}\ \square_{i=1}^n (b_i \to gc_i)\ [\Gamma \mid \Theta]$ is simply the process for the alternation as defined above.



Diag. 17.



Diag. 18.

Loops

Let $\mathbf{a} : \mathbf{A} \longrightarrow \mathbf{U}$ be the interpretation of the alternation in $\mathsf{do} \, [\!]_{i=1}^{n} (b_i \rightarrow gc_i) \; [\Gamma \mid \Theta]$. Recall that the intended semantics of do is that the loop exits when none of the guards are satisfiable. Define *exit* to be the program which exits when all the other guards fail:

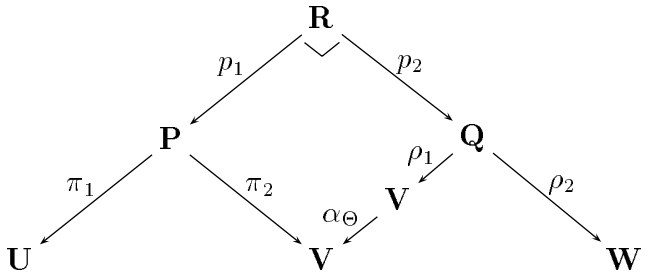
$$exit = \neg(\bigwedge_{1=i}^{n} b_i) \rightarrow \mathsf{nil} \; [\Gamma \mid \Theta]$$

Using the scheme outlined above, this is interpreted by the system $\mathbf{e} = \tau_{\mathbf{E},\mathbf{U}} : \mathbf{E} \longrightarrow \mathbf{U}$ having a single unobserved transition which can be taken precisely when the other guards fail.

Assume $\mathbf{A} = (J, A)$ and $\mathbf{E} = (K, E)$. To interpret the do loop requires combining $\mathbf{A}$ and $\mathbf{E}$ in such a way that the loop is closed but can exit when the alternation fails. This is achieved by identifying the start and end points of the alternation $\mathbf{A}$ (closing the loop) and the start point of $\mathbf{E}$. This yields a system $\mathbf{P} = (L, P)$ and a morphism $\mathbf{p} : \mathbf{P} \longrightarrow \mathbf{U}$. The shape L is the colimit on the left in Diagram 18. The functor P and morphism $\mathbf{p}$ are constructed following the same pattern as for sequential composition.

Parallel composition

Let $gc_1 \; [\Gamma_1 \mid \Theta_1, \Theta]$ and $gc_2 \; [\Gamma_2 \mid \Theta, \Theta_2]$ be programs interpreted by $\mathbf{p} : \mathbf{P} \longrightarrow \mathbf{U} \times \mathbf{V}$ and $\mathbf{q} : \mathbf{Q} \longrightarrow \mathbf{V} \times \mathbf{W}$ in which $\mathbf{V}$ is the system for the shared interface $\Theta = [c_1 : \mathbb{X}_1, \ldots, c_n : \mathbb{X}_n]$. To form the system for $\mathsf{chan} \; \Theta \; \mathsf{in} \; gc_1 \parallel gc_2$, we first project the common interface from $\mathbf{p}$ and $\mathbf{q}$. Before pulling back, the input/output roles for one process are reversed for each channel in Θ using α_Θ defined earlier (Diagram 19).



Diag. 19.

The common interface is hidden to yield the morphism:

$$\mathbf{r} = \langle \pi_1 \circ p_1, \rho_2 \circ p_2 \rangle : \mathbf{R} \longrightarrow \mathbf{U} \times \mathbf{W}$$

9 Discussion

We have described a generalization of transition systems with shapes labelled in a category (rather than an alphabet) via the Grothendieck construction. With a suitable universe of shapes, the theory accommodates both concurrency and asynchrony and allows states and actions to have more structure than in conventional models.

The main contribution of the paper is a framework for typed processes with interaction and communication via limits in categories of twisted systems. The theory has been applied to giving models of a simple imperative language with message passing primitives. We have elected to use reflexive graphs for shapes but anticipate that the semantics can be adapted to cubical sets.

The categorical semantics for the language can be related to the transition relation generated by a conventional structural operational semantics as follows. Let $\mathbf{P} = (J, P)$ be a system and construct the (reflexive) *graph of elements*, $\oint \mathbf{P}$, as follows. Vertices are pairs (a, x) where a is a vertex in J and $x \in P\widetilde{a}$. An edge $(a, x) \longrightarrow (b, y)$ is a pair (e, z) where $e : a \longrightarrow b$ is an edge in J and $z \in Pe$ such that $(Pe_1)(z) = x$ and $(Pe_2)(z) = y$. This construction extends to a functor such that given $\mathbf{p} = (f, \eta) : \mathbf{P} \longrightarrow \mathbf{U}$, then $(\oint \mathbf{p})(a, x) = (fa, \eta_a x)$. This is a variation on the discrete Conduché fibration discovered by Lamarche for functors $\widetilde{\mathsf{J}} \longrightarrow \mathsf{Set}$ where $\widetilde{\mathsf{J}}$ is the twisted arrow category of J. See [18,8,7,32].

The connection with a conventional operational semantics appears when we interpret the vertices of $\oint \mathbf{P}$ as states consisting of a program point and a variable assignment at that point. By definition, there is a transition $(a, x) \longrightarrow (b, y)$ in $\oint \mathbf{P}$ precisely when the program can evolve from a to b and when the span associated with the edge $e : a \longrightarrow b$ relates x to y.

Recall that for the language given earlier, the shape of the interface $\mathbf{U}$ has only one vertex and that vertex is always labelled by the terminal. It follows that $\oint \mathbf{U}$ has only a single vertex and that $\oint \mathbf{p} : \oint \mathbf{P} \to \oint \mathbf{U}$ simply labels the

edges of $\oint \mathbf{P}$. Thus, $\oint \mathbf{p}$ resembles a conventional labelled transition system. The construction of $\oint \mathbf{p}$ is also similar to the expansion of value-passing CCS into basic CCS.

With respect to equivalences on processes, starting from labelled reflexive graphs such as $\oint \mathbf{p}$ constructed above, one can either adapt the standard definition of bisimulation for conventional transition systems or use the characterization of bisimulation in terms of open maps due to Joyal et al. [34]. Alternatively, one can define equivalences on $\mathbf{p}$ directly without expanding to the graph of elements. These topics are addressed in a forthcoming paper.

We intend to investigate other modes of communication from that described here. Interaction here is point-to-point through a channel shared by exactly two processes. This differs from CCS (and other languages) where the synchronization of transitions from two processes does not preclude those transitions from synchronizing with transitions in other processes. In a related vein, we would like to investigate encoding name-passing/mobility of the sort found in the π-calculus [37] by representing channel identifiers as data.

Finally, the parallel language is clearly less expressive than what can be accommodated by the theory, particularly with respect to process types. Channels in the language are interpreted by systems with shapes restricted to graphs with one vertex and a single non-identity transition. We would prefer protocols in which one could specify that a channel carries data of different sorts according to its state and where each transition in a channel is assigned a relative direction of flow of information. This requires developing a syntax and theory for describing types and expressing when a process has a particular type.

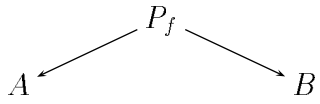
Acknowledgement

My thanks to Francois Lamarche, Till Plewe, Steve Vickers and Krzysztof Worytkiewicz for both their technical assistance and encouragement. I am especially grateful to Krzysztof for our many discussions which have helped greatly in my understanding of the material presented here. Thanks also to the anonymous referees for their thorough reading and comments on an earlier draft.

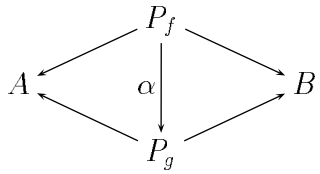
A Spans, twists and oplaxness

The axioms for *bicategories* are similar to those for 2-categories except that identity and associativity axioms hold up to isomorphism rather than equality and are subject to coherence conditions. For details see Borceux [5] or Bénabou [4]. One can quotient the 1-cells of a bicategory to obtain a category. Following Bénabou, the category obtained by identifying all 1-cells which are 2-isomorphic is the *classifying category*. We write $\mathsf{B}^\flat$ for the classifying category for a bicategory B.

Given a category C with pullbacks, the *bicategory of spans*, $\mathsf{Sp}(\mathsf{C})$, has as 0-cells the objects of C. A 1-cell $f : A \longrightarrow B$ is a *span*; a diagram in C of the form:



A 2-cell $\alpha : f \Longrightarrow g$ is a morphism in C such that the two triangles commute:



Composition of 1-cells is by pullback.

Following Carboni et al. [10], a morphism $f : A \longrightarrow B$ in a bicategory B is a *map* when it has a right adjoint $f^* : B \longrightarrow A$. A morphism $f : A \longrightarrow B$ in $\mathsf{Sp}(\mathsf{C})$ is a map if and only if the left leg of f is an isomorphism in C.

Let $F, G : \mathsf{J} \longrightarrow \mathsf{Sp}(\mathsf{C})$ be functors. An *oplax natural transformation* $\alpha : F \Longrightarrow\!\!\!\!\circ\; G$ consists of the following data:

(i) for every object a in J a morphism $\alpha_a : Fa \longrightarrow Ga$ in $\mathsf{Sp}(\mathsf{C})$.

(ii) for each pair of objects $a, b \in \mathsf{J}$ a natural transformation:

$$\tau_{ab} : c_{Fa,Fb,Gb}(-, \alpha_b) \circ F_{ab} \Longrightarrow c_{Fa,Ga,Gb}(\alpha_a, -) \circ G_{ab}$$

where $c_{abc} : \mathsf{Sp}(\mathsf{C})(a,b) \times \mathsf{Sp}(\mathsf{C})(b,c) \longrightarrow \mathsf{Sp}(\mathsf{C})(a,c)$ is the composition bifunctor and $c_{Fa,Ga,Gb}(\alpha_a, -)$ and $c_{Fa,Fb,Gb}(-, \alpha_b)$ are the functors obtained by fixing α_a or α_b.

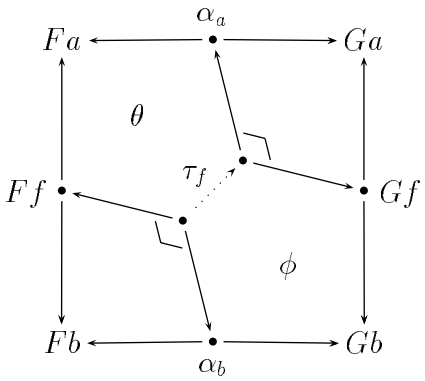
There are coherence conditions which we omit.

Diagram 20 shows oplax naturality for $f : a \longrightarrow b$ in J where τ_f is the f component of the natural transformation τ_{ab}. The definition translates to the requirement that the pentagons θ and ϕ must commute. The difference between this and a naturality diagram is simply that in the latter τ_f is the identity.

If J is a reflexive graph, then write $\mathsf{OplaxMap}(FJ, \mathsf{Sp}(\mathsf{C}))$ for the bicategory with objects functors $FJ \longrightarrow \mathsf{Sp}(\mathsf{C})$ and morphisms oplax natural transformations whose 1-cell components are maps. The correspondence between twists and spans referred to earlier in the paper is made precise in the following theorem.

Theorem A.1 *The quotient functor category* $\mathsf{OplaxMap}^{\flat}(FJ, \mathsf{Sp}(\mathsf{C}))$ *is isomorphic to* $\mathsf{C}^{F\tilde{J}}$.

The proof is a corollary of a theorem in [18].



Diag. 20.

References

[1] S. Abramsky, S. J. Gay, and R. Nagarajan. Interaction categories and foundations of typed concurrent programming. In M. Broy, editor, *Deductive Program Design: Proceedings of the 1994 Marktoberdorf International Summer School*, NATO ASI Series F: Computer and Systems Sciences. Springer-Verlag, 1995.

[2] Samson Abramsky. Interaction Categories (Extended Abstract). In G. L. Burn, Simon J. Gay, and M. D. Ryan, editors, *Theory and Formal Methods 1993: Proceedings of the First Imperial College Department of Computing Workshop on Theory and Formal Methods*, pages 57–70. Springer-Verlag Workshops in Computer Science, 1993.

[3] Samson Abramsky. Interaction Categories and communicating sequential processes. In A. W. Roscoe, editor, *A Classical Mind: Essays in Honour of C. A. R. Hoare*, pages 1–15. Prentice Hall International, 1994.

[4] Jean Bénabou. Introduction to bicategories. *Lecture Notes in Mathematics*, 47, 1967.

[5] Francis Borceux. *Handbook of Categorical Algebra 1, Basic Category Theory.* Encyclopedia of Mathematics and its Applications. Cambridge University Press, 1994.

[6] S.D. Brookes. Full abstraction for a shared variable parallel language. In *Proceedings of the IEEE Symposium on Logic in Computer Science.* IEEE Computer Society Press, 1993.

[7] M.C. Bunge and M.P. Fiore. Unique factorization lifting functors and categories of processes. preprint, October 1998.

[8] M.C. Bunge and S.B. Niefield. Exponentiability and single universes. *Journal of Pure and Applied Algebra*, To appear.

[9] Rod Burstall. An algebraic description of programs with assertions, verification and simulation. In J. Mack Adams, John Johnston, and Richard Stark, editors, *Conference on Proving Assertions about Programs*, pages 7–14. ACM, 1972.

[10] Aurelio Carboni, Stefano Kasangian, and Ross Street. Bicategories of spans and relations. *Journal of Pure and Applied Algebra*, 33:259–267, 1984.

[11] Gian Luca Cattani and Vladimiro Sassone. Higher dimensional transition systems. In *Proceedings of the IEEE Symposium on Logic in Computer Science*, pages 55–62. IEEE Computer Society Press, 1996.

[12] Gian Luca Cattani, Ian Stark, and Glynn Winskel. A presheaf semantics for the π-calculus. Technical Report RS–97–34, BRICS, 1997.

[13] David Clark, Lindsay Errington, and Chris Hankin. Static analysis of value-passing process calculi (extended abstract). In *Proceedings of the 1994 Theory and Formal Methods Section Workshop*. Imperial College Press, 1995.

[14] J. R. B. Cockett and D. A. Spooner. SProc categorically. In *CONCUR 94: Fifth International Conference on Concurrency*, Lecture Notes in Computer Science, Uppsala, Sweden, 1994. Springer-Verlag.

[15] J. R. B. Cockett and D. A. Spooner. Categories for synchrony and asynchrony. *Electronic Notes in Computer Science*, 1, 1995.

[16] J. R. B. Cockett and D. A. Spooner. Constructing process categories. 1995.

[17] Roy Crole. *Categories for Types*. Cambridge University Press, 1993.

[18] Lindsay Errington. *Twisted Systems*. PhD thesis, Imperial College, Submitted.

[19] Lindsay Errington. Categories of processes with state. In *Third Theory and Formal Methods Workshop*. IC Press, 1996.

[20] Lindsay Errington. Twisted systems and the logic of imperative programs. preprint, October 1998.

[21] G. Ferrari. *Unifying Models of Concurrency*. PhD thesis, University Pisa, 1990.

[22] Gianluigi Ferrari, Ugo Montanari, and Miranda Mowbray. Structured transition systems with parametric observations: observational congruences and minimal realizations. *Mathematical Structures in Computer Science*, 1996.

[23] M. Fiore, E. Moggi, and D. Sangiorgi. A fully abstract model for the π-calculus. In *Proceedings of the IEEE Symposium on Logic in Computer Science*. IEEE Computer Society Press, 1996.

[24] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[25] Joseph A. Goguen. A categorical manifesto. *Mathematical Structures in Computer Science*, 1(1):49–67, 1991.

[26] Joseph A. Goguen. Sheaf semantics for concurrent interacting objects. *Mathematical Structures in Computer Science*, 2:159–191, 1992.

[27] Eric Goubault. *Géométrie du Parallélisme*. PhD thesis, École Polytechnique, 1995.

[28] Eric Goubault and Thomas Jensen. Homology of higher-dimensional automata. In *CONCUR '92*, Lecture Notes in Computer Science. Springer Verlag, 1992.

[29] J.W. Gray. Fibred and cofibred categories. In S. Eilenberg, D.K. Harrison, S. MacLane, and H. Röhrl, editors, *Conference on Categorical Algebra*, pages 21–83. Springer Verlag, 1966.

[30] M. Hennessy and A. Ingólfsdóttir. A theory of communicating processes with value passing. *Information and Computation*, 107(2), 1993.

[31] Anna Ingólfsdóttir. A semantic theory for value passing processes late approach, Parts I and II. Technical Report RS–95–3/22, BRICS, 1995.

[32] Peter Johnstone. A note on discrete Conduché fibrations. *Theory and Applications of Categories*, 5(1):1–11, 1999.

[33] A. Joyal, R. Street, and D. Verity. Traced monoidal categories. *Math. Proc. Camb. Phil. Soc.*, 119:447–468, 1996.

[34] André Joyal, Mogens Nielsen, and Glynn Winskel. Bisimulation and open maps. In *Proceedings of the Eighth IEEE Symposium on Logic in Computer Science*. IEEE Computer Society Press, 1993.

[35] David May. *Occam 2 language definition*. INMOS Ltd., 1987.

[36] José Meseguer and Ugo Montanari. Petri nets are monoids. *Information and Computation*, 88:105–155, 1990.

[37] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, Parts I and II. *Information and Computation*, 100:1–77, 1992.

[38] A. M. Pitts. Categorical logic. Technical Report 367, University of Cambridge Computer Laboratory, May 1995. 94 pages.

[39] V.R. Pratt. Modeling concurrency with geometry. In *Proc. 18th Ann. ACM Symposium on Principles of Programming Languages*, pages 311–322, January 1991.

[40] Ian Stark. A fully abstract domain model for the π-calculus. In *Proceedings of the IEEE Symposium on Logic in Computer Science*. IEEE Computer Society Press, 1996.

[41] Andrzej Tarlecki, Rod Burstall, and Joseph Goguen. Some fundamental algebraic tools for the semantics of computation, part 3: Indexed categories. *Theoretical Computer Science*, 91:239–264, 1991.

[42] G. Winskel. Event Structures. In *Advances in Petri Nets 1986, Part II*, volume 255 of *Lecture Notes in Computer Science*. Springer Verlag, 1987.

[43] G. Winskel and M. Nielsen. Models for concurrency. In S. Abramsky, D. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 4, pages 1–148. Oxford University Press, 1995.

[44] Glynn Winskel. Synchronisation trees. *Theoretical Computer Science*, 34:33–82, 1984.

[45] Glynn Winskel. An introduction to event structures. In *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, page 364. REX Workshop, 1988. Springer Verlag LNCS 354.

[46] Glynn Winskel. A presheaf semantics of value-passing processes. Technical Report RS–96–44, BRICS, 1996.